A Guide to Time Series Analysis in Python

Time series analysis is a common task for data scientists. This guide will introduce you to its key concepts in Python.

Written bySadrach Pierre

Image: Shutterstock

UPDATED BY
Rose Velazquez | Oct 12, 2022
REVIEWED BY
Parul Pandey | Oct 11, 2022

Across industries, organizations commonly use time series data, which means any information collected over a regular interval of time, in their operations. Examples include daily stock prices, energy consumption rates, social media engagement metrics and retail demand, among others. Analyzing time series data yields insights like trends, seasonal patterns and forecasts into future events that can help generate profits. For example, by understanding the seasonal trends in demand for retail products, companies can plan promotions to maximize sales throughout the year.

When analyzing time series data, you should undertake a number of steps. First, you need to check for stationarity and autocorrelation. Stationarity is a way to measure if the data has structural patterns like seasonal trends. Autocorrelation occurs when future values in a time series linearly depend on past values. You need to check for both of these in time series data because they're assumptions that are made by many widely used methods in time series analysis. For example, the autoregressive integrated moving average (ARIMA) method for forecasting time series assumes stationarity. Further, linear regression for time series forecasting assumes that the data has no autocorrelation. Before conducting these processes, then, you need to know if the data is viable for the analysis.

During a time series analysis in Python, you also need to perform trend decomposition and forecast future values. Decomposition allows you to visualize trends in your data, which is a great way to clearly explain their behavior. Finally, forecasting allows you to anticipate future events that can aid in decision making. You can use many different techniques for time series forecasting, but here, we will discuss the autoregressive integrated moving average (ARIMA).

We will be working with publicly available airline passenger time series data, which can be found here.

TIME SERIES ANALYSIS IN PYTHON

Across industries, organizations commonly use time series data, which means any information collected over a regular interval of time, in their operations. Examples include daily stock prices, energy consumption rates, social media engagement metrics and retail demand, among others. Analyzing time series data yields insights like trends, seasonal patterns and forecasts into future events that can help generate profits. For example, by understanding the seasonal trends in demand for retail products, companies can plan promotions to maximize sales throughout the year.

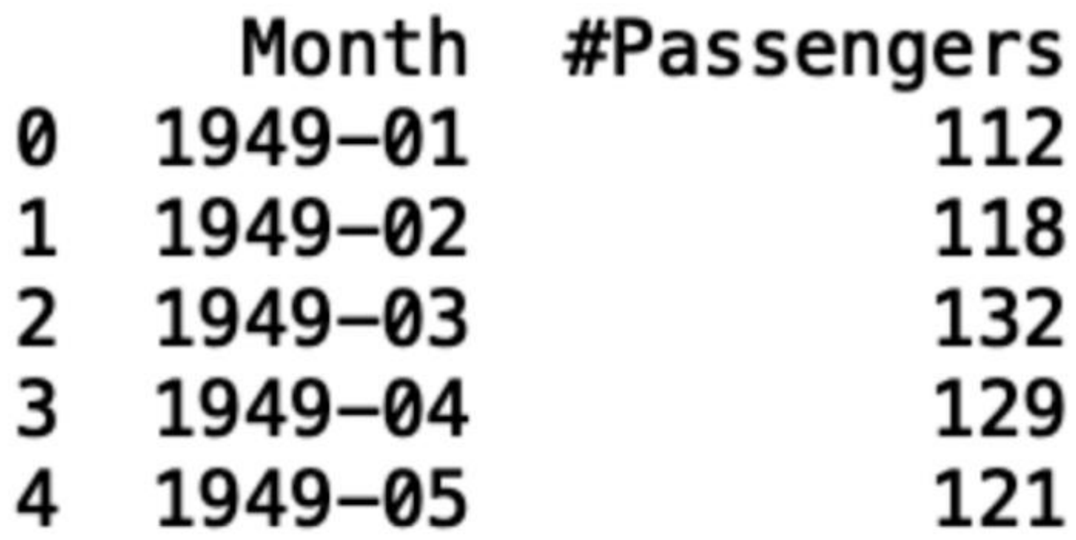
Reading and Displaying Data

To start, let’s import the Pandas library and read the airline passenger data into a data frame:

```
import pandas as pd
df = pd.read_csv("AirPassengers.csv")
```

Now, let’s display the first five rows of data using the data frame head() method:

```
print(df.head())
```

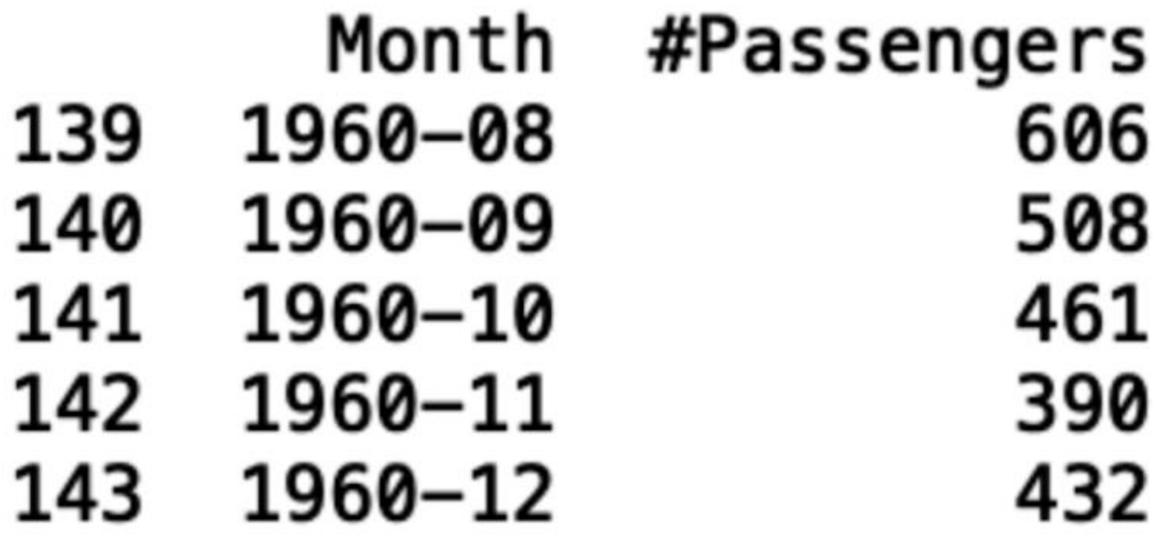


```
     Month  #Passengers
0  1949-01          112
1  1949-02          118
2  1949-03          132
3  1949-04          129
4  1949-05          121
```

We can see that the data contains a column labeled “Month” that contains dates. In that column, the dates are formatted as year–month. We also see that the data starts in the year 1949.

The second column is labeled “#Passengers,” and it contains the number of passengers for the year–month. Let’s take a look at the last five records the data using the tail() method:

```
print(df.tail())
```

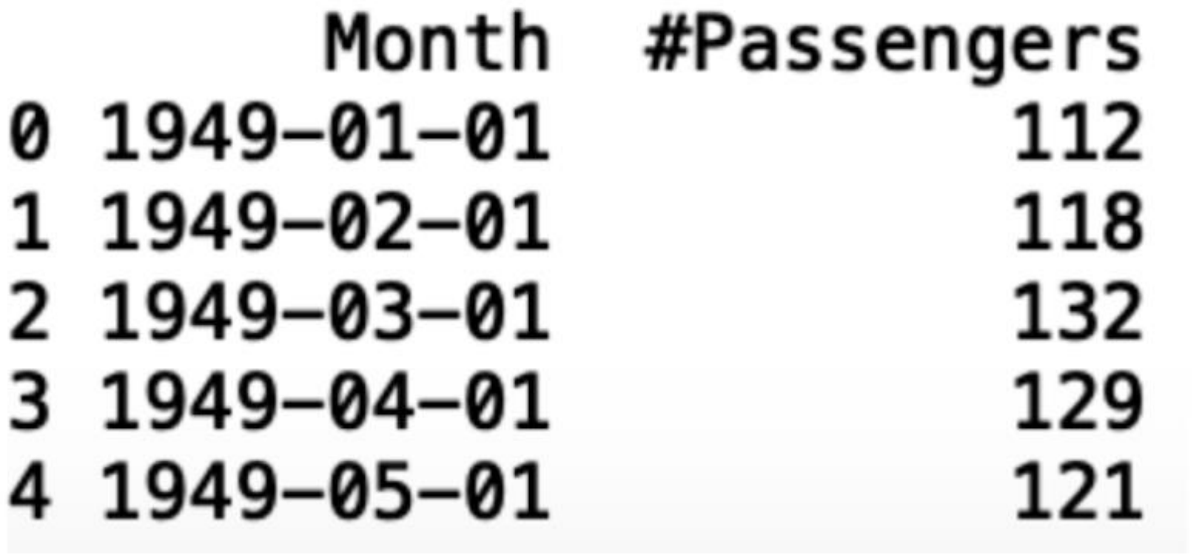


```
       Month  #Passengers
139  1960-08          606
140  1960-09          508
141  1960-10          461
142  1960-11          390
143  1960-12          432
```

We see from this process that the data ends in 1960.

The next thing we will want to do is convert the month column into a datetime object. This will allow it to programmatically pull time values like the year or month for each record. To do this, we use the Pandas to_datetime() method:

```
df['Month'] = pd.to_datetime(df['Month'], format='%Y-%m')
print(df.head())
```

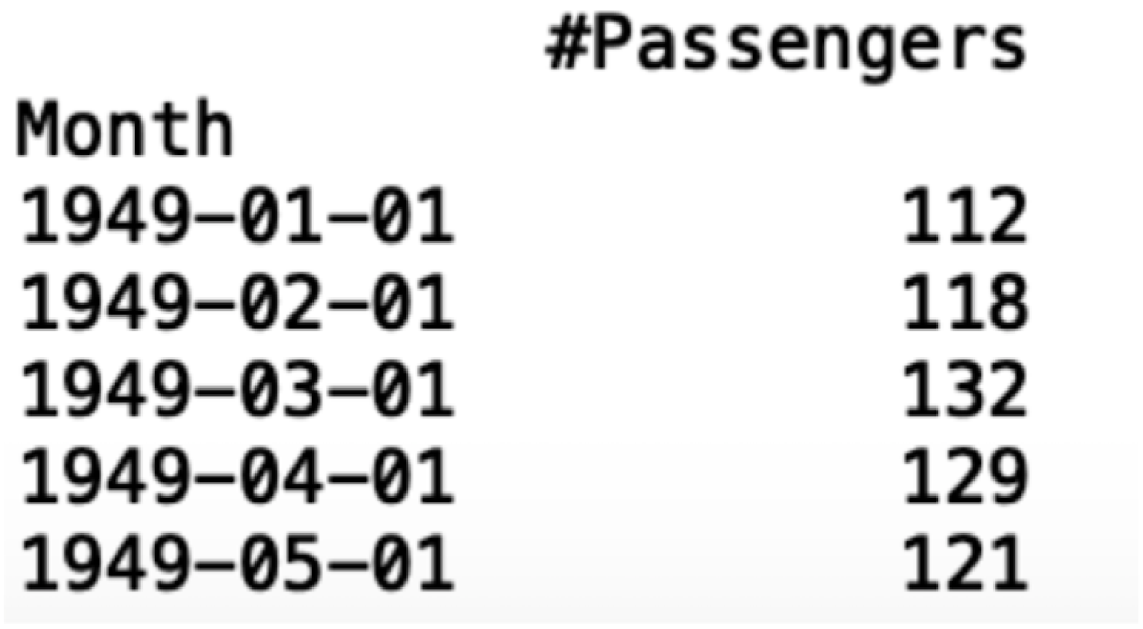


```
       Month  #Passengers
0 1949-01-01          112
1 1949-02-01          118
2 1949-03-01          132
3 1949-04-01          129
4 1949-05-01          121
```

Note that this process automatically inserts the first day of each month, which is basically a dummy value since we have no daily passenger data.

The next thing we can do is convert the month column to an index. This will allow us to more easily work with some of the packages we will be covering later:

```
df.index = df['Month']
del df['Month']
print(df.head())
```



```
            #Passengers
Month
1949-01-01          112
1949-02-01          118
1949-03-01          132
1949-04-01          129
1949-05-01          121
```

Image: Screenshot

Next, let’s generate a time series plot using Seaborn and Matplotlib. This will allow us to visualize the time series data. First, let’s import Matplotlib and Seaborn:

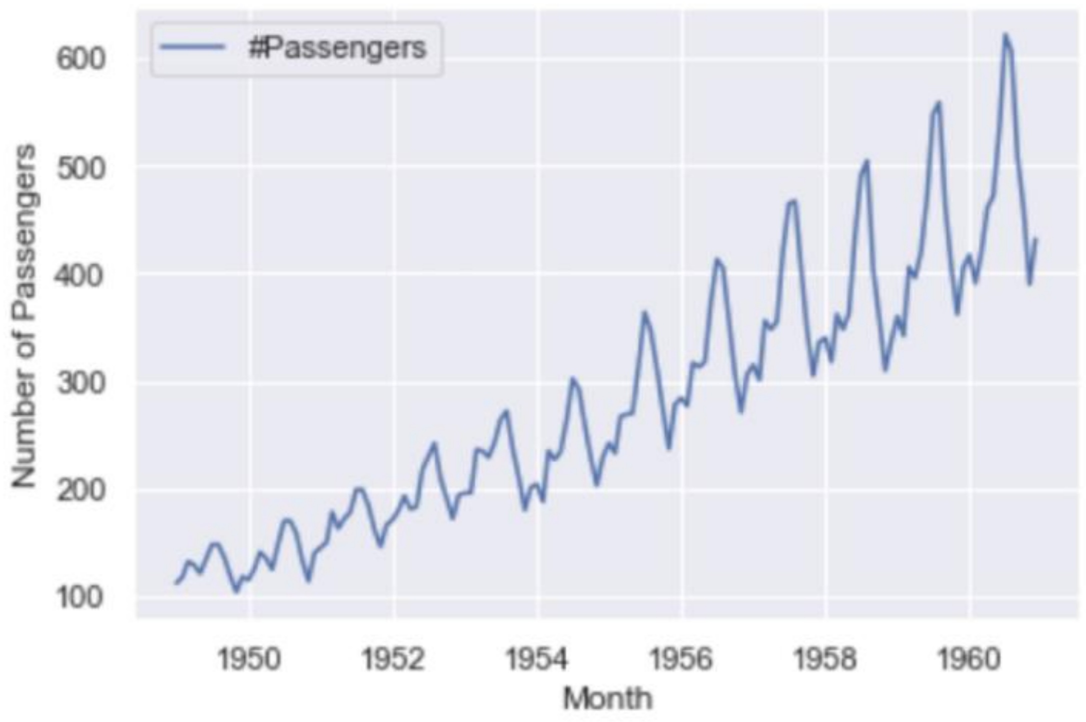
```
import matplotlib.pyplot as plt
import seaborn as sns
```

Next, let's generate a line plot using Seaborn:

```
sns.lineplot(df)
```

And label the y-axis with Matplotlib:

```
plt.ylabel("Number of Passengers")
```



MORE FROM SADRACH PIERRE Need to Perform Financial Data Analysis? Why Python Is Your Best Tool.

Stationarity

Stationarity is a key part of time series analysis. Simply put, stationarity means that the manner in which time series data changes is constant. A stationary time series will not have any trends or seasonal patterns. You should check for stationarity because it not only makes modeling time series easier, but it is an underlying assumption in many time series methods. Specifically, stationarity is assumed for a wide variety of time series forecasting methods including autoregressive moving average (ARMA), ARIMA and Seasonal ARIMA (SARIMA).

We will use the Dickey Fuller test to check for stationarity in our data. This test will generate critical values and a p-value, which will allow us to accept or reject the null hypothesis that there is no stationarity. If we reject the null hypothesis, that means we accept the alternative, which states that there is stationarity.

These values allow us to test the degree to which present values change with past values. If there is no stationarity in the data set, a change in present values will not cause a significant change in past values.

Let's test for stationarity in our airline passenger data. To start, let's calculate a seven-month rolling mean:

```
rolling_mean = df.rolling(7).mean()
rolling_std = df.rolling(7).std()
```

Next, let's overlay our time series with the seven-month rolling mean and seven-month rolling standard deviation. First, let's make a Matplotlib plot of our time series:

```
plt.plot(df, color="blue",label="Original Passenger Data")
```

Then the rolling mean:

```
plt.plot(rolling_mean, color="red", label="Rolling Mean Passenger Number")
```

And finally, the rolling standard deviation:

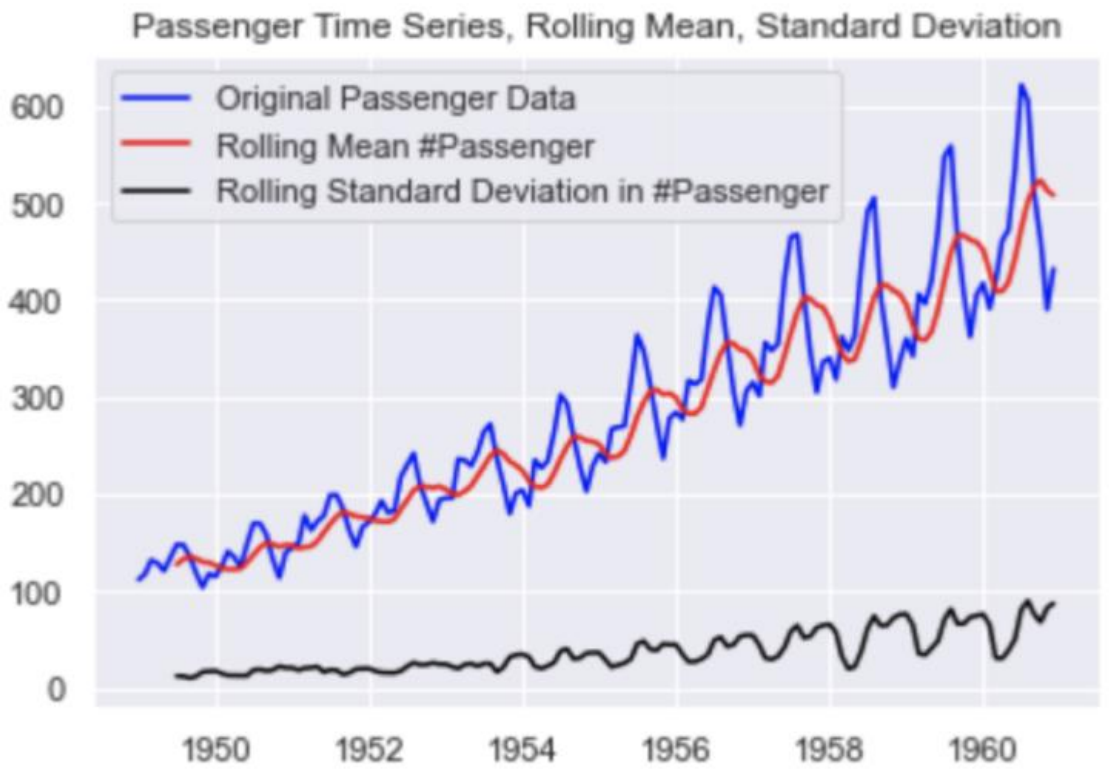
```
plt.plot(rolling_std, color="black", label = "Rolling Standard Deviation
in Passenger Number")
```

Let’s then add a title:

```
plt.title("Passenger Time Series, Rolling Mean, Standard Deviation")
```

And a legend:

```
plt.legend(loc="best")
```



Next, let’s import the augmented Dickey-Fuller test from the statsmodels package. The documentation for the test can be found here.

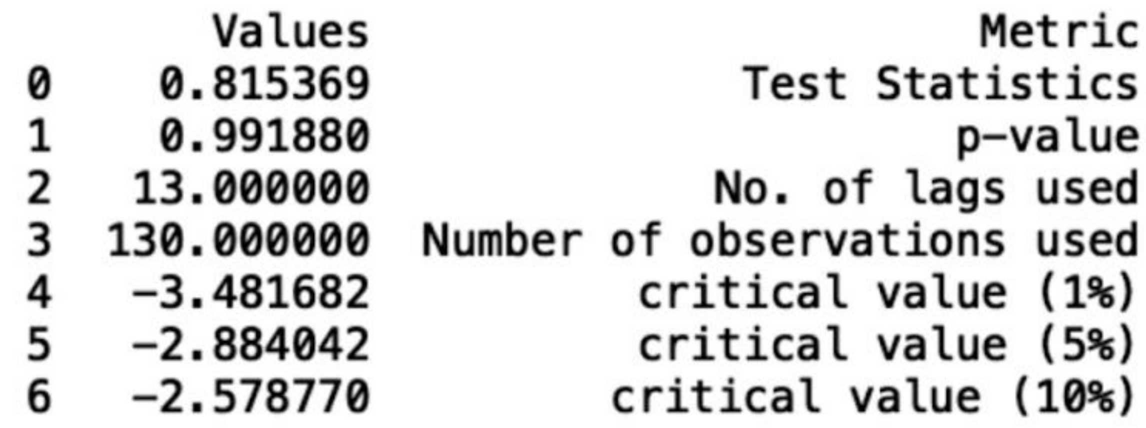
```
from statsmodels.tsa.stattools import adfuller
```

Next, let's pass our data frame into the adfuller method. Here, we specify the autolag parameter as "AIC," which means that the lag is chosen to minimize the information criterion:

```
adft = adfuller(df,autolag="AIC")
```

Next, let's store our results in a data frame display it:

```
output_df = pd.DataFrame({"Values":[adft[0],adft[1],adft[2],adft[3],
adft[4]['1%'], adft[4]['5%'], adft[4]['10%']]  , "Metric":["Test
Statistics","p-value","No. of lags used","Number of observations used",
                                                         "critical value
(1%)", "critical value (5%)", "critical value (10%)"]})
print(output_df)
```



```
         Values                       Metric
0      0.815369              Test Statistics
1      0.991880                      p-value
2     13.000000             No. of lags used
3    130.000000  Number of observations used
4     -3.481682          critical value (1%)
5     -2.884042          critical value (5%)
6     -2.578770         critical value (10%)
```

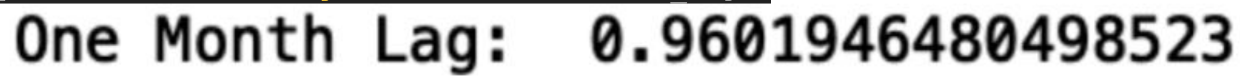
We can see that our data is not stationary from the fact that our p-value is greater than 5 percent and the test statistic is greater than the critical value. We can also draw these conclusions from inspecting the data, as we see a clear, increasing trend in the number of passengers.

Autocorrelation

Checking time series data for autocorrelation in Python is another important part of the analytic process. This is a measure of how correlated time series data is at a given point in time with past values, which has huge implications across many industries. For example, if our passenger data has strong autocorrelation, we can assume that high passenger numbers today suggest a strong likelihood that they will be high tomorrow as well.

The Pandas data frame has an autocorrelation method that we can use to calculate the autocorrelation in our passenger data. Let’s do this for a one-month lag:

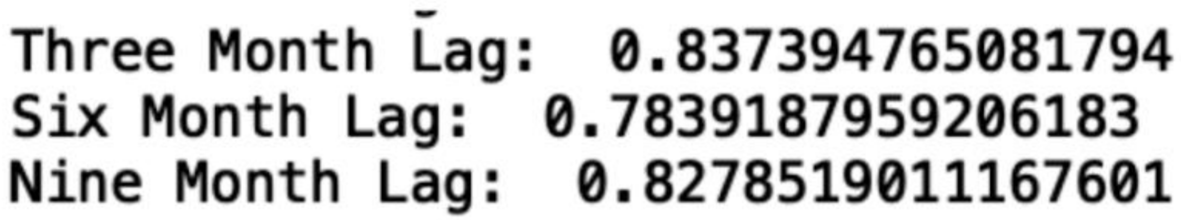
```
autocorrelation_lag1 = df['#Passengers'].autocorr(lag=1)

print("One Month Lag: ", autocorrelation_lag1)
```



```
One Month Lag:  0.9601946480498523
```

Now, let’s try three, six and nine months:

```
autocorrelation_lag3 = df['#Passengers'].autocorr(lag=3)

print("Three Month Lag: ", autocorrelation_lag3)


autocorrelation_lag6 = df['#Passengers'].autocorr(lag=6)

print("Six Month Lag: ", autocorrelation_lag6)


autocorrelation_lag9 = df['#Passengers'].autocorr(lag=9)

print("Nine Month Lag: ", autocorrelation_lag9)
```



```
Three Month Lag:  0.837394765081794
Six Month Lag:  0.783918795920618З
Nine Month Lag:  0.8278519011167601
```

We see that, even with a nine-month lag, the data is highly autocorrelated. This is further illustration of the short- and long-term trends in the data.

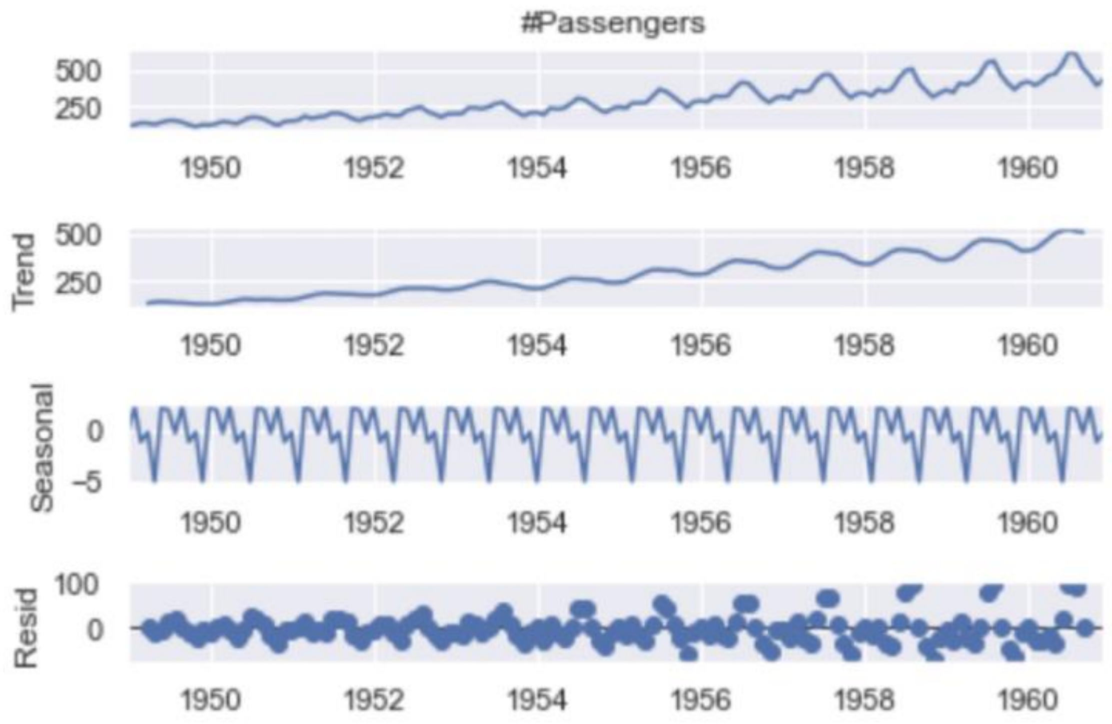
Decomposition

Trend decomposition is another useful way to visualize the trends in time series data. To proceed, let's import seasonal_decompose from the statsmodels package:

```
from statsmodels.tsa.seasonal import seasonal_decompose
```

Next, let's pass our data frame into the seasonal_decompose method and plot the result:

```
decompose = seasonal_decompose(df['#Passengers'],model='additive',
period=7)
decompose.plot()
plt.show()
```



From this plot, we can clearly see the increasing trend in number of passengers and the seasonality patterns in the rise and fall in values each year.

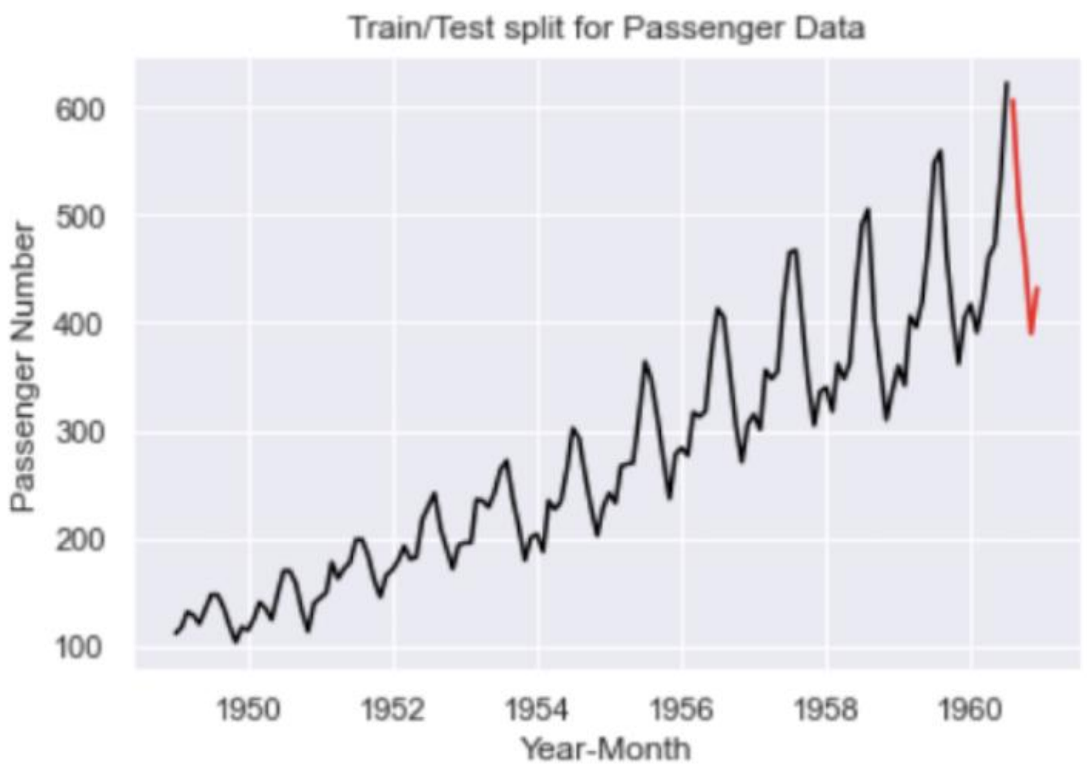
Forecasting

Time series forecasting allows us to predict future values in a time series given current and past data. Here, we will use the ARIMA method to forecast the number of passengers, which allows us to forecast future values in terms of a linear combination of past values. We will use the auto_arima package, which will allow us to forgo the time consuming process of hyperparameter tuning.

First, let's split our data for training and testing and visualize the split:

```
df['Date'] = df.index
```

```
train = df[df['Date'] < pd.to_datetime("1960-08", format='%Y-%m')]

train['train'] = train['#Passengers']

del train['Date']

del train['#Passengers']

test = df[df['Date'] >= pd.to_datetime("1960-08", format='%Y-%m')]

del test['Date']

test['test'] = test['#Passengers']

del test['#Passengers']

plt.plot(train, color = "black")

plt.plot(test, color = "red")

plt.title("Train/Test split for Passenger Data")

plt.ylabel("Passenger Number")

plt.xlabel('Year-Month')

sns.set()

plt.show()
```

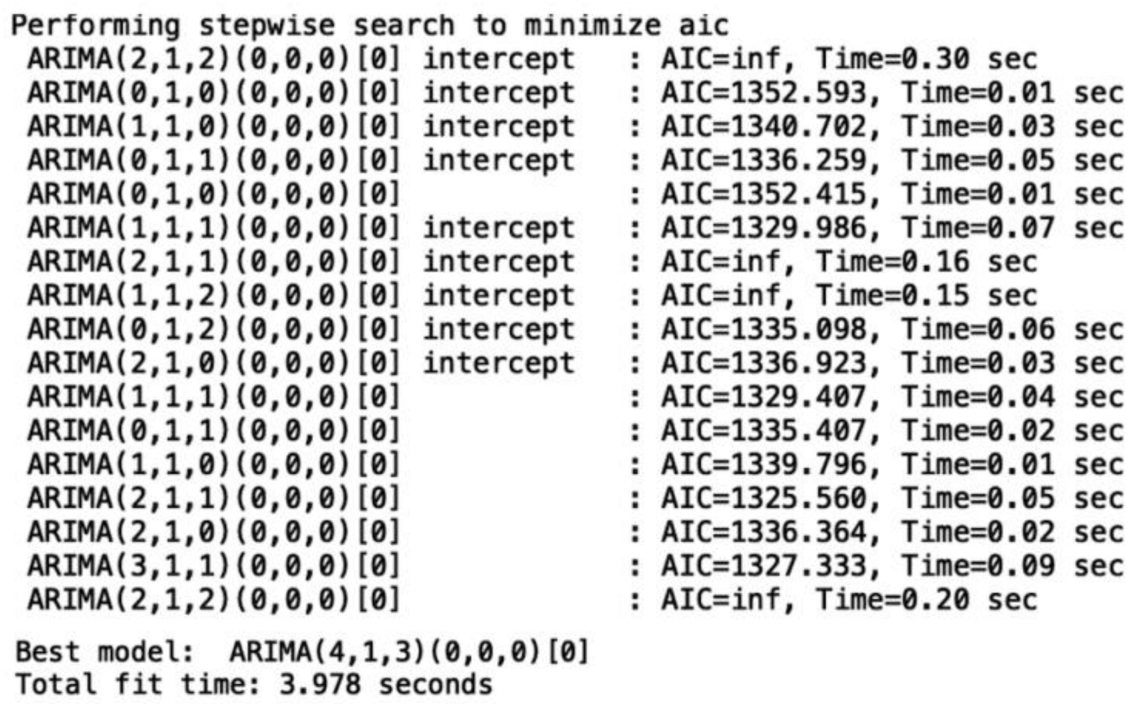


The black line corresponds to our training data and the red line corresponds to our test data.

Let's import auto_arima from the pdmarima package, train our model and generate predictions:

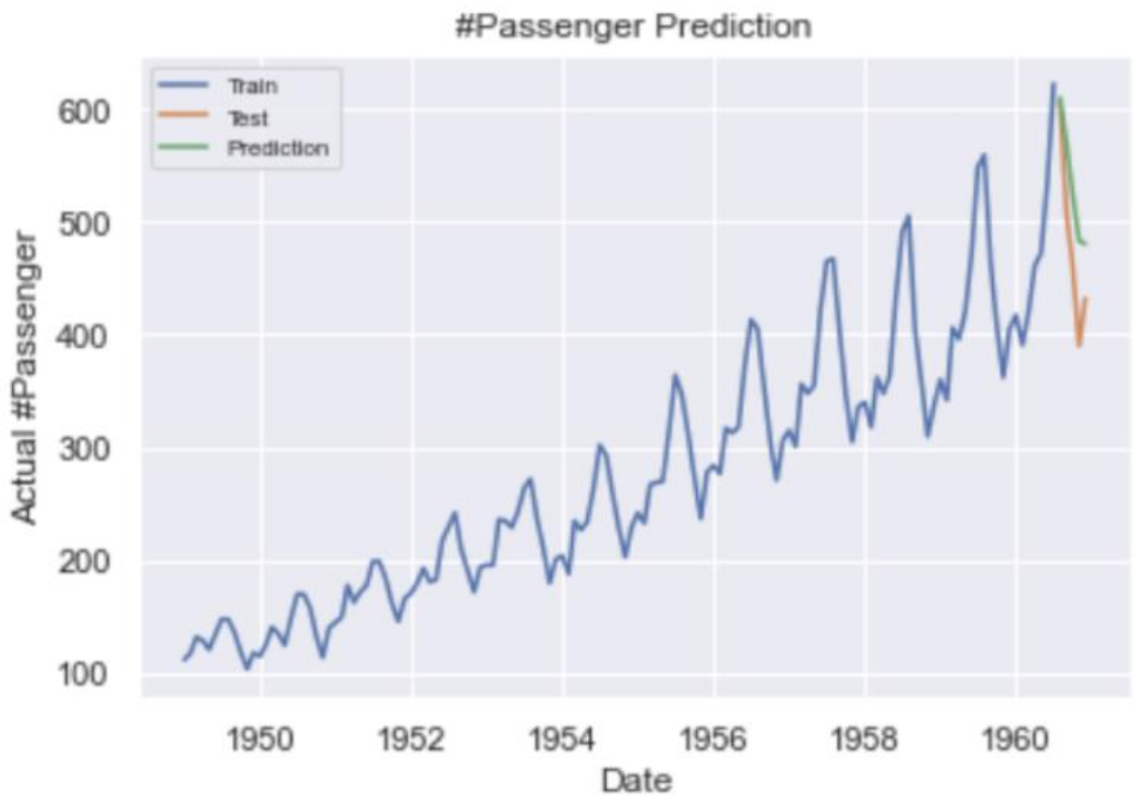
```
from pmdarima.arima import auto_arima

model = auto_arima(train, trace=True, error_action='ignore',

suppress_warnings=True)

model.fit(train)

forecast = model.predict(n_periods=len(test))

forecast = pd.DataFrame(forecast,index =

test.index,columns=['Prediction'])
```

Below is a truncated sample of the output:



```
Performing stepwise search to minimize aic
 ARIMA(2,1,2)(0,0,0)[0] intercept   : AIC=inf, Time=0.30 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=1352.593, Time=0.01 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=1340.702, Time=0.03 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=1336.259, Time=0.05 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=1352.415, Time=0.01 sec
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=1329.986, Time=0.07 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : AIC=inf, Time=0.16 sec
 ARIMA(1,1,2)(0,0,0)[0] intercept   : AIC=inf, Time=0.15 sec
 ARIMA(0,1,2)(0,0,0)[0] intercept   : AIC=1335.098, Time=0.06 sec
 ARIMA(2,1,0)(0,0,0)[0] intercept   : AIC=1336.923, Time=0.03 sec
 ARIMA(1,1,1)(0,0,0)[0]             : AIC=1329.407, Time=0.04 sec
 ARIMA(0,1,1)(0,0,0)[0]             : AIC=1335.407, Time=0.02 sec
 ARIMA(1,1,0)(0,0,0)[0]             : AIC=1339.796, Time=0.01 sec
 ARIMA(2,1,1)(0,0,0)[0]             : AIC=1325.560, Time=0.05 sec
 ARIMA(2,1,0)(0,0,0)[0]             : AIC=1336.364, Time=0.02 sec
 ARIMA(3,1,1)(0,0,0)[0]             : AIC=1327.333, Time=0.09 sec
 ARIMA(2,1,2)(0,0,0)[0]             : AIC=inf, Time=0.20 sec

Best model:  ARIMA(4,1,3)(0,0,0)[0]
Total fit time: 3.978 seconds
```

Now, let's display the output of our model:



Our predictions are shown in green and the actual values are shown in orange.

Finally, let's calculate root mean squared error (RMSE):

```
from math import sqrt
from sklearn.metrics import mean_squared_error
rms = sqrt(mean_squared_error(test,forecast))
print("RMSE: ", rms)
```

```
RMSE:  61.36535942376535
```

MASTER DATA SCIENCE The Ultimate Guide to ROC Curves and AUC

Importance of Time Series Analysis in Python

Conducting time series data analysis is a task that almost every data scientist will face in their career. Having a good understanding of the tools and methods for analysis can enable data scientists to uncover trends, anticipate events and consequently inform decision making. Understanding the seasonality patterns through stationarity, autocorrelation and trend decomposition can guide promotion planning throughout the year, which can improve profits for companies. Finally, time series forecasting is a powerful way to anticipate future events in your time series data, which can also significantly impact decision making. These types of analyses are invaluable to any data scientist or data science team that looks to bring value to their company with time series data. The code from this post is available on GitHub.